

ABSTRACT

Title of thesis: USING MANY-CORE COMPUTING
TO SPEED UP DE NOVO
TRANSCRIPTOME ASSEMBLY

Sean O'Brien, Master of Science, 2016

Thesis directed by: Professor Uzi Vishkin
University of Maryland Institute for
Advanced Computer Studies and
Department of Electrical and Computer
Engineering

The central dogma of molecular biology implies that DNA holds the blueprint which determines an organism's structure and functioning. However, this blueprint can be read in different ways to accommodate various needs, depending on a cell's location in the body, its environment, or other external factors. This is accomplished by first transcribing DNA into messenger RNA (mRNA), and then translating mRNA into proteins. The cell regulates how much each gene is transcribed into mRNA, and even which parts of each gene is transcribed. A single gene may be transcribed in different ways by splicing out different parts of the sequence. Thus, one gene may be transcribed into many different mRNA sequences, and eventually into different proteins.

The set of mRNA sequences found in a cell is known as its transcriptome, and it differs between tissues and with time. The transcriptome gives a biologist a snapshot of the cell's state, and can help them track the progression

of disease, etc. Some modern methods of transcriptome sequencing give only short reads of the mRNA, up to 100 nucleotides. In order to reconstruct the mRNA sequences, one must use an assembly algorithm to stitch these short reads back into full length transcripts.

De novo transcriptome assemblers are an important family of transcriptome assemblers. Such assemblers reconstruct the transcriptome without using a reference genome to align to and are, therefore, computationally intensive. We present here a de novo transcriptome assembler designed for a parallel computer architecture, the XMT architecture. With this assembler we produce speedups over existing de novo transcriptome assemblers without sacrificing performance on traditional quality metrics.

USING MANY-CORE COMPUTING TO SPEED UP
DE NOVO TRANSCRIPTOME ASSEMBLY

by

Sean O'Brien

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2016

Advisory Committee:

Professor Uzi Vishkin, Chair/Advisor

Professor Hector Corrada Bravo

Professor Yavuz Oruc

© Copyright by
Sean O'Brien
2016

To Julia, for putting up with me.

Acknowledgments

I would like to express my deep appreciation to Uzi Vishkin, for his help and guidance throughout my graduate education. Thanks also to James Edwards, for answering countless questions with limitless patience. I am grateful to the NCI-UMD Partnership for Integrative Cancer Research for their support, and to Javed Khan and his group at National Cancer Institute for their helpful comments. I would also like to thank my thesis committee, Hector Corrada Bravo and Yavuz Oruc for their support and advice. Finally, thank you to my friends and family, especially Max Morawski, Isaac Carruthers, Natasha Hagemeyer and Sadie Bickley, for being there for me at all hours.

Table of Contents

List of Tables	vi
List of Figures	vii
1 Introduction	1
2 Background	4
2.1 The Transcriptome and the Role of Messenger RNA	4
2.1.1 Transcription	5
2.1.1.1 Splicing	5
2.1.2 Translation	7
2.2 Transcriptome Sequencing and Assembly	7
2.2.1 Transcriptome Sequencing	9
2.2.2 Transcriptome Assembly	10
2.2.2.1 De Bruijn Graph Assembly Algorithms	11
2.2.2.2 Existing De Novo Transcriptome Assemblers	13
2.3 The Explicit Multi-Threading (XMT) Architecture	13
3 Methods	15
3.1 Algorithm for Parallel De Novo Transcriptome Assembly	15
3.1.1 K-mer Counting and De Bruijn Graph Construction	15
3.1.1.1 Count (k/2)-mers	17
3.1.1.2 Group k-mers By Their First Half	18
3.1.1.3 Count k-mers Within Each Group	18
3.1.1.4 Compact Graph	19
3.1.2 Greedy Contig Construction	20
3.1.3 Evaluation of Connections Between Contigs	22
3.1.4 Graph Simplification	23
3.1.4.1 Short Tip Removal	24
3.1.4.2 Bubble Merging	24
3.1.5 Reconstruction of Full Length Transcripts	25
3.1.5.1 Read Mapping and Loop Resolution	29

3.1.5.2	Paired Read Extension	30
3.1.5.3	Paired Read Grouping	31
3.1.5.4	Transcript Building	33
3.2	Evaluation of the XMT Assembler	35
3.2.1	Datasets	35
3.2.2	Assemblers	36
3.2.3	Evaluation Metrics	37
3.2.3.1	Run Time and Memory Use	37
3.2.3.2	Assembly Quality	38
4	Results	40
4.1	Assembly Quality	40
4.2	Run Time and Memory Use	42
4.3	Runtime Scaling with Processor Count	42
4.4	Parallelism During Greedy Contig Construction	46
5	Conclusions	50
	Bibliography	51

List of Tables

4.1	Transcript Length Statistics	41
4.2	Assembly Quality Statistics	43
4.3	Total Runtime and Peak Memory Usage	45

List of Figures

2.1	DNA Structure	6
2.2	Transcription and translation	6
2.3	Alternative splicing	8
2.4	Sequencing and assembly	8
2.5	De Bruijn graph	12
3.1	Overview of the XMT Assembler algorithm	16
3.2	Graph compaction	26
3.3	Bubbles and short tips	26
3.4	Typical structures of connected components	27
3.5	Paired read extension	32
4.1	Evaluation metric comparison	44
4.2	Speedups over serial assemblers	47
4.3	Total run time vs # TCUs used	47
4.4	Effect of using more threads during contig construction	49

Chapter 1: Introduction

In cell biology messenger RNA (mRNA) is an important class of molecules which control the cell's production of proteins. mRNA molecules are copies of the cell's DNA, genes which have been modified and prepared for translation into the final protein product [10]. The set of mRNA molecules in a cell is known as its transcriptome, and unlike the genome, it can vary drastically between cells and change over time as mRNA is produced by transcribing DNA and consumed after translation into protein. Biologists are often interested in the contents of the transcriptome because it gives further insight into the expression of genes than the genome.

Next Generation Sequencing (NGS) are DNA and RNA sequencing techniques that have been developed and used extensively in the past decades [22]. While NGS has made huge improvements on sequencing throughput over past techniques, most NGS techniques read only short sequences of DNA or RNA at a time. As a result transcriptome assembly, the computational problems of reconstructing the transcriptome based on the sequencing reads, is more difficult [18]. While many transcriptome assemblers rely on using a reference genome to align to, there is a class of assemblers called de novo assemblers

which use only the reads produced by assembly.

De novo transcriptome assemblers are generally better than genome-guided transcriptome assemblers at detecting novel transcripts, and are especially important for species with an incomplete reference genome. However, de novo transcriptome assemblers generally consume much more time than genome-guided assemblers [17].

In this paper, we present the XMT Assembler, a de novo transcriptome assembler designed for the Explicit Multi-Threaded (XMT) parallel architecture. We chose the XMT architecture over other parallel architectures because it is designed from the ground up to support parallel random access model (PRAM) programming [27]. This includes concurrent access of all processors to a shared memory and efficient support of fine-grained, irregular parallelism. Due to the nature of de novo transcriptome assembly algorithms, these advantages are important for effectively exploiting parallelism. By leveraging these advantages, we are able to achieve significant speedups over the fastest serial de novo transcriptome assemblers while maintaining traditional assembly quality metrics.

While the current project did not seek implementation on commodity platforms, we expect that the parallelism exposed by the current work can be a first step towards improved performance on such platforms. For exploiting this parallelism for performance gains, this parallelism will need to be coarsened and the coarsening will need to be done with minimal overheads. But, will such coarsening be possible? Though the data can be divided naturally into

localized partitions in later stages, no such natural divisions exist in early stages. Alternatively, commodity systems may choose in the future to upgrade their support for fine-grained irregular parallelism to bring their capabilities closer to XMT. Given the quest to support deep learning applications, the case for such an upgrade seems to gain broadening support.

In chapter 2 of this paper, we describe in more detail the role of mRNA, the techniques used to analyze the transcriptome, and features of the XMT architecture. In chapter 3, we describe in detail the algorithm for the XMT Assembler and how we evaluated the performance of the XMT Assembler. In chapter 4, we present the results of our evaluation, and in chapter 5 we discuss our final conclusions.

Chapter 2: Background

2.1 The Transcriptome and the Role of Messenger RNA

DNA holds the genetic information of a cell, based on which all the cell's proteins are built. This information is encoded in the sequence of nucleotides which comprise the DNA molecule. There are four basic types nucleotides in DNA, and the order of these nucleotides in a gene determines which amino acids the resulting protein will have. These nucleotides are adenine (A), cytosine (C), guanine (G), and thymine (T). Each type of nucleotide has a “complementary” nucleotide type with which it can bond. Adenine bonds with thymine and guanine pairs with cytosine. The the complete DNA molecule consists of two strands of nucleotides with complementary sequences [28]. See figure 2.1 on page 6.

Messenger RNA (mRNA) is the set of RNA molecules which transfer genetic information from DNA to the ribosomes, where they are then translated into proteins (see figure 2.2 on 6). DNA is the long-term storage of the cell and the information it stores does not vary much between cells in an organism. Changes in protein production are instead achieved by regulating which

genes are transcribed into precursor mRNA (pre-mRNA), and by regulating how the pre-mRNA is processed into mature mRNA. An mRNA molecule is called a transcript, and the set of all mRNA molecules in a cell is called its transcriptome.

2.1.1 Transcription

Transcription is the process of producing mRNA from DNA. A protein called RNA polymerase binds to a particular region in the DNA molecule known as a promoter region. The RNA polymerase then traverses the DNA, using the DNA sequence as the template to add matching RNA nucleotides to a pre-mRNA molecule. The level of expression of a particular gene is regulated by many factors, including changes to the physical structure of the DNA molecule, methylation of the DNA, and general transcription factors which position RNA polymerase at the promoter region [\[24\]](#).

2.1.1.1 Splicing

After pre-mRNA is copied from the DNA template, it undergoes further processing to convert it to mature mRNA. One major change is splicing, where certain sections of the mRNA, called introns, are removed. The remaining sections, called exons, are placed next to each other. Different copies of a single pre-mRNA sequence may be spliced in many different ways via alternative splicing, where different sets of exons are retained for each mature mRNA

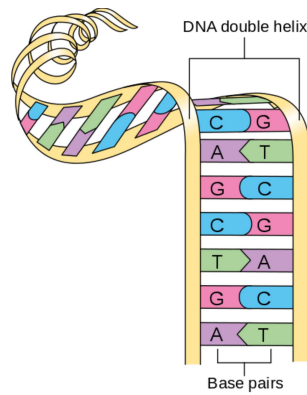


Figure 2.1: Sketch of a DNA molecule from Cancer Research UK [2], showing the complementary arrangement of the nucleotides on its two strands.

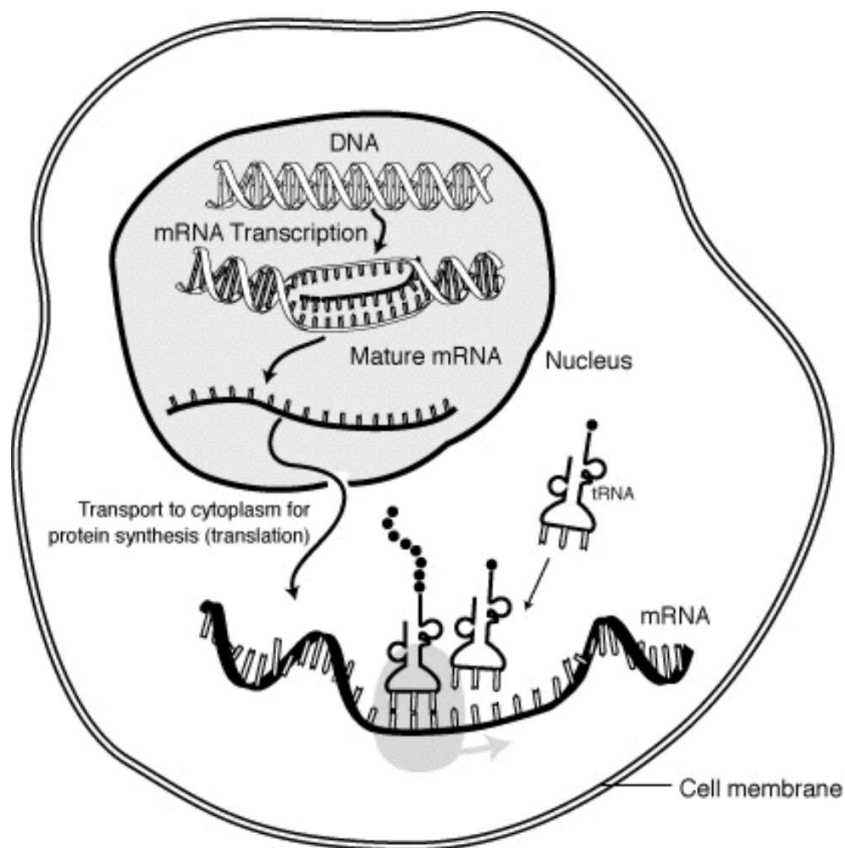


Figure 2.2: Overview of the transcription and translation process from the National Institutes of Health [19], showing how mRNA is used in converting DNA into protein.

molecule. In this way, a single gene can code for many different proteins [8].

See figure 2.3 on page 8.

2.1.2 Translation

Translation is the process of producing proteins from mRNA. The cellular machinery used to accomplish this is called ribosome. A ribosome is composed of proteins and ribosomal RNA molecules, which are distinct from mRNA. The ribosome binds an mRNA molecule and begins traversing the molecule. For each set of three nucleotides (called a codon), the ribosome adds one amino acid to a growing protein. This continues until the ribosome reaches a “stop codon”, which signals that translation is complete, and the ribosome releases the protein and mRNA.

2.2 Transcriptome Sequencing and Assembly

In order to analyse the cellular transcriptome, we would like to create a digital representation of the transcript sequences. This is accomplished in two steps. First we sequence the transcriptome, producing digital “reads” of small sections of each transcript. Then we assemble the original full length transcript sequences using an assembly algorithm. See figure 2.4 on page 8.

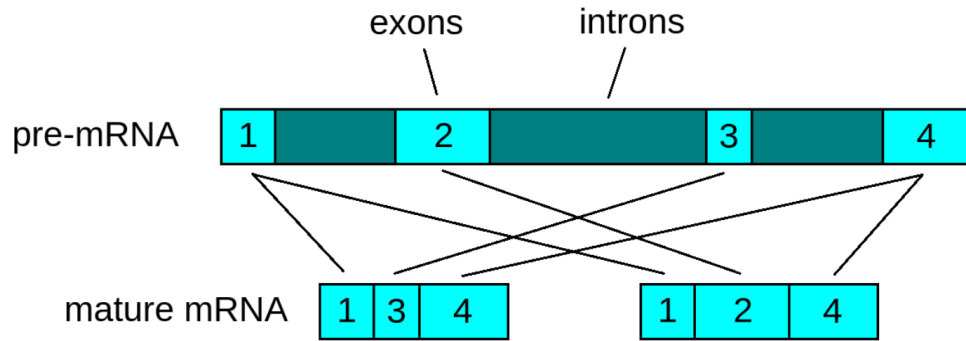


Figure 2.3: Different copies of the same pre-mRNA can be spliced in many ways to produce different mature mRNA sequences and eventually different proteins.

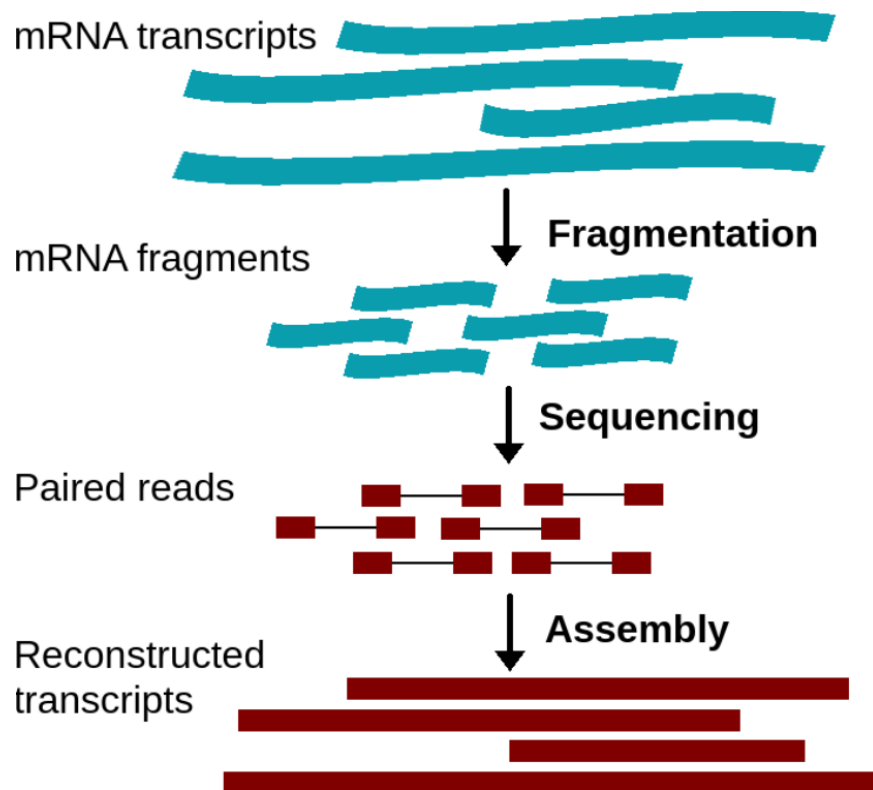


Figure 2.4: Overview of the process of analyzing physical mRNA transcripts to discover their nucleotide sequence. Blue shapes represent physical mRNA molecules and red shapes represent digital nucleotide sequences.

2.2.1 Transcriptome Sequencing

There are several modern sequencing methods. We will discuss here the most widely used method, short-read RNA-seq. This is the method used by the popular Illumina platform [1], and the method upon which our assembly algorithm is based.

In short-read RNA-seq, mRNA is fragmented into “fragments” of hundreds of nucleotides. These fragments are converted into complementary DNA (cDNA) fragments via reverse transcription. With the Illumina platform, one end of each cDNA fragment is fixed to a surface and cloned repeatedly, creating one cluster of fragments for each original fragment. Then, nucleotides marked with a fluorescent dye are bonded to the end of the cDNA fragments one at a time, and the dye color is measured after each nucleotide is added. Using this process, a shorter section of the cDNA fragment (typically around 100 nucleotides) is sequenced. This sequence is called a “read”.

After sequencing one end of the cDNA fragment, the fragment is cloned again, but with the other end of the fragment fixed to the surface. Sequencing the fragment again produces another read, giving the sequence of the other end of the cDNA fragment. Together with the first read, these are called a “paired read”. The final output of transcriptome sequencing is a set of paired reads.

2.2.2 Transcriptome Assembly

Transcriptome assembly is the computation problem of taking the paired reads produced by sequencing and reconstructing the sequence of the original transcriptome. There are two basic assembly methods: genome-guided and de novo assembly. In genome-guided assembly, reads are aligned to a reference genome and transcripts are reconstructed based on overlaps between reads in this alignment. In de novo assembly, no reference genome is used, so the assembler uses only the set of reads to reconstruct the transcriptome. As a result, the assembly is not biased toward any reference and is better at identifying structural changes such as alternative splicing. The assembly algorithm we present in this paper is a de novo assembler.

There are two major classes of algorithms for de novo transcriptome assembly: overlap layout consensus algorithm and de Bruijn graph algorithms [16]. In overlap layout consensus algorithms, every set of two reads is checked for overlap. An overlap graph is built where each read is a vertex, and an edge is drawn between two reads if they overlap. This graph is then simplified and evaluated to produce full-length transcripts. Overlap layout consensus algorithms are not well-suited for assembling modern sequencing data which produce many short reads. This is because these algorithms are computationally intensive, requiring checking every set of two reads for overlap, and thus quadratic work for construction of the overlap graph.

2.2.2.1 De Bruijn Graph Assembly Algorithms

De Bruijn graph assembly algorithms improve the computational efficiency by building a special overlap graph called a de Bruijn graph. In a de Bruijn graph each vertex corresponds to a unique sequence of length $k-1$, called a $(k-1)$ -mer. An edge may be drawn between two vertices if they overlap by $k-2$ nucleotides. Though only a $k-2$ overlap is required to draw an edge between two vertices, each edge corresponds to a unique sequence of length k , called a k -mer. This k -mer is the $(k-1)$ -mer of the predecessor edge with the last nucleotide of the successor edge appended to it.

In de Bruijn graph assembly algorithms, a de Bruijn graph is built based on the k -mers and $(k-1)$ -mers found in the set of reads. The number of times each k -mer and $(k-1)$ -mer appears in the read set is counted, and edges and vertices are added for each unique k -mer and $(k-1)$ -mer. This standardizes the sequence length of vertices in the overlap graph, and the length of overlap between vertices which correspond to edges. As a result, the graph can be built with linear work, instead of quadratic work as in overlap layout consensus algorithms.

After a de Bruijn graph is constructed from the read set, it can be converted into a compacted de Bruijn graph by merging certain vertices which have only one successor with that successor. The resulting vertex will represent a longer sequence, incorporating all of the merged $(k-1)$ -mers. All the edges in the graph, however, still represent k -mers. See figure 2.5 on page 12.

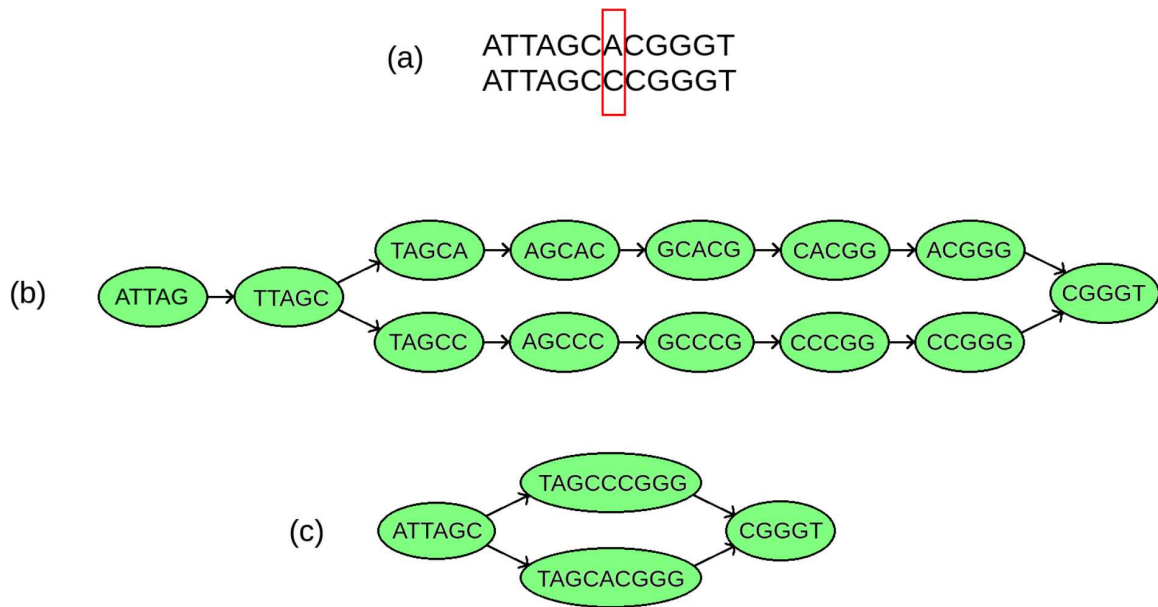


Figure 2.5: Given the reads in (a), which differ in only one nucleotide, and a k -mer size of 6, the de Bruijn graph in (b) will be constructed. Note that the vertices represent $(k-1)$ -mers, and the edges represent k -mers. After compaction, the graph in (c) is produced. While the vertices now represent longer sequences, the edges still represent k -mers.

2.2.2.2 Existing De Novo Transcriptome Assemblers

There are several existing de novo transcriptome assemblers, such Trinity [7], Oases [25], and Soapdenovo-Trans [29]. These examples are all de Bruijn graph assemblers, using typical k-mer sizes of between 20 and 40 nucleotides. The parallel assembler we present in this text adapts many of the techniques used by these assemblers for parallel computation. Our assembler most closely resembles the Trinity assembler at a high level of abstraction, but borrows some heuristics from the other assemblers where they are more suitable for parallel implementation.

There are de novo transcriptome assemblers which have been adapted to run in parallel on a standard multi-core system with shared memory, such as Trinity [9], as well as de novo transcriptome assemblers designed for distributed systems, such as Trans-ABYSS [23, 26]. The XMT Assembler is the only assembler designed for a many-core architecture with thousands of processors.

2.3 The Explicit Multi-Threading (XMT) Architecture

The XMT architecture is a many-core computer architecture which aims to improve single-task completion time and ease-of-programming for fine-grained parallel applications by supporting Parallel Random Access Model (PRAM) programming [27]. PRAM is the foremost model for parallel algorithms [11, 14] in the theory of computer science and algorithms. We will discuss two par-

allel programming features which XMT supports, the spawn block and the prefix-sum operation.

A spawn block denotes a region of parallel code. When a spawn block is encountered, many threads are created, each of which executes at its own pace. Each thread is assigned to a small processor called a thread control unit (TCU). The TCU then executes the code within the spawn block. If there are more threads than TCUs, then threads are assigned to TCUs as the TCUs become available. The spawn block is complete only when all threads have been processed, at which point the program can continue.

Prefix-sum is an operation that can be performed within a spawn block. This operation takes two arguments, a base variable and an increment variable. After the prefix-sum is operation, the base variable is incremented by the value of the increment variable, and the increment variable is set to the original value of the base variable. This is an atomic operation, meaning it completes in one step from the point of view of other threads. The prefix-sum operation can be used to synchronize access to shared data within a spawn block. The XMT architecture has dedicated prefix-sum hardware which allows for fast multi-operand prefix-sum calculation when multiple threads perform the prefix-sum operation at the same time.

Chapter 3: Methods

3.1 Algorithm for Parallel De Novo Transcriptome Assembly

Our parallel de novo transcriptome assembler (the XMT Assembler) uses the de Bruijn graph framework used by Trinity, SOAPdenovo-Trans, and Oases. Specifically, it is based on the Trinity assembler and thus follows the same general steps as the Trinity assembler, though sometimes the order of computation is changed to facilitate parallelization. The XMT Assembler has five major steps, as outlined in figure [3.1](#) on [16](#)

3.1.1 K-mer Counting and De Bruijn Graph Construction

The XMT assembler can use any choice of k-mer size less than or equal to 32. This limit is due to the space required to store a k-mer. We currently use two 32-bit words to store each k-mer, and since each base requires 2 bits to store, 32 bases is the maximum that can be stored using this scheme. For

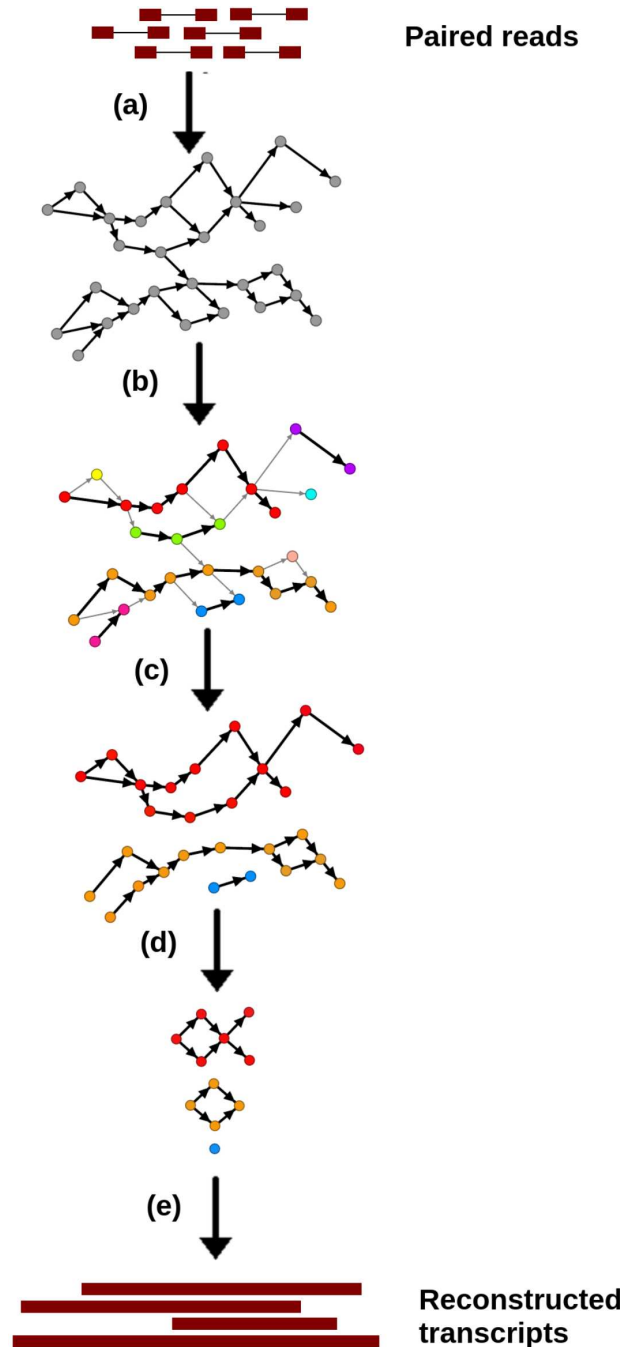


Figure 3.1: An overview of the XMT Assembler algorithm. The input to the algorithm is the set of paired reads produced by transcriptome sequencing. The algorithm (a) counts k -mers and builds a de Bruijn graph (b) finds non-overlapping linear “contigs” in a greedy manner (c) evaluates the edges between contigs and removes those that are not well supported (d) simplifies the graph by removing structures that are likely the result of sequencing errors, and (e) finds paths through the de Bruijn graph that are likely to represent true transcripts.

testing, we chose a k-mer size of 25, the default k-mer size for the Trinity assembler. This choice was arbitrary, but close to the default k-mer size of other assemblers. For example, the default k-mer size of SOAPdenovo-Trans is 23.

The k-mer counting stage is the first stage of the assembly algorithm. Thus, the input to this stage is the output of the sequencer, or sequence simulator. The XMT assembler accepts strand-specific paired reads in the FASTA format. “Strand-specific” means that we know which read is closer to the start of transcript, and which end is closer to the end of the transcript. The algorithm could be modified to accept unstranded paired reads. This would require counting two k-mers for each k-mer found in the reads: the unmodified k-mer and the reverse complement of the k-mer. A perfect assembly using unstranded reads would therefore produce two copies of each transcript: one forward copy and one reverse complement copy.

The XMT assembler counts k-mers and (k-1)-mers in three stages, counting (k/2)-mers, grouping k-mers by their first half, and then counting k-mers within each group. The k-mer counting stage of the algorithm is based on work done by James Edwards [4].

3.1.1.1 Count (k/2)-mers

An array is created with an entry for each (k/2)-mer. Each entry will track the number of times its (k/2)-mer is found as the first half of a k-mer

in the reads. Since there are four possible values for each base, this requires $4^{\lfloor k/2 \rfloor}$ entries. Our default value of k is 25, and we round $k/2$ down to 12, so 16 million entries are required. One thread traverses each read, using the prefix-sum operation to count each $(k/2)$ -mer found.

3.1.1.2 Group k -mers By Their First Half

The prefix-sum of the count array from the previous step is performed. This allows us to group the k -mers by their first half by giving the index where the first element which starts with each $(k/2)$ -mer must be stored. Then each read is traversed again- this time the entire k -mer is inspected and copied to an array, using the $(k/2)$ -mer index and the prefix-sum operation so that k -mers are grouped by $(k/2)$ -mer in the array, but unsorted within each $(k/2)$ -mer group.

3.1.1.3 Count k -mers Within Each Group

Small Groups:

Each group smaller than some threshold is processed serially in its own thread. The k -mers in the group are sorted, and then each k -mer and $(k-1)$ -mer is counted. When a new k -mer or $(k-1)$ -mer is found, the prefix-sum operation is used to assign an index for a new edge or vertex, respectively. The vertices corresponding to each edge are assigned by inspecting the two $(k-1)$ -mers contained by the edge's k -mer.

Large Groups:

Each large group is processed in parallel, using a similar method as was used to count the first half of each k-mer. Another array of $4^{\lceil k/2 \rceil}$ elements is used to hold the count of each k-mer within the group. A thread is assigned to each entry in the group. It finds the correct index in the array based on the second half of its k-mer. If this is the first time this k-mer has been found, a new edge is assigned to represent this k-mer. Otherwise, the count of the representative edge is incremented using the prefix-sum operation. Meanwhile, (k-1)-mers are also counted using an array of $4^{\lceil k/2 \rceil - 1}$ elements.

3.1.1.4 Compact Graph

Once all k-mers and (k-1)-mers have been counted and assigned to edges and vertices, the “light” graph is copied to a more memory intensive “full” graph, which includes vertices that hold pointers to “dna vectors”, a data structure which can hold an unlimited number of bases, still storing each base using 2 bits.

The de Bruijn graph is then compacted. The algorithm identifies “linear” vertices: vertices which have only one successor and which are the only predecessor of their successor. These vertices can be merged with their successor, taking its outgoing edges. It also identifies all linear vertices which are the start of a linear sequence as the representative vertex for that sequence. A thread is assigned to each representative vertex, and the subsequent vertices

are merged into it, adding their bases and count statistics to it. Finally, the representative vertex takes the edges of the first vertex it finds which is not “linear”. When this is complete, the vertices no longer correspond to $(k-1)$ -mers, but can hold longer sequences. The edges, however, still correspond to unique k -mers and there is still a $k-2$ overlap between all adjacent vertices. See figure 3.2 on page 26.

3.1.2 Greedy Contig Construction

In this stage we find non-overlapping contigs in the de Bruijn graph. A contig in the context of the de Bruijn graph is a simple chain of vertices. Each chain of vertices represents a sequence of nucleotides, and since the chains do not overlap, no two contigs share any $(k-1)$ -mers. The goal of this stage is to divide the graph into as few contigs as possible and to minimize the number of edges which connect two different contigs. Edges which connect different contigs are the subject of the next stage, and are known as “welds”.

Our approach is based on Trinity’s Inchworm stage, which performs the same function. In the Trinity algorithm, the k -mers are sorted in descending order of count in the read set. The most abundant k -mer is used as a seed and a contig is built by choosing the next k -mer with $(k-1)$ -mer overlap with the highest count. Though a de Bruijn graph has not yet been built at this stage of the Trinity algorithm, this is equivalent to choosing the vertex with the highest count, and then extending by choosing the successor with the highest

count. K-mers are added until one is reached which has no successor which has not been used in a previous contig. The contig is then extended by looking at predecessors in a similar manner.

In order to facilitate parallelism, our algorithm runs in two passes. First, each thread chooses a random vertex, without regard for vertex count. This vertex becomes the seed for a draft contig, using the same algorithm as Trinity's Inchworm. During the construction of this draft contig, the vertex with the highest count is recorded. Then, during the second pass, this vertex is used as a seed. This allows us to find a sort of local maximum.

The problem with choosing all the vertices with the maximum counts as seeds is that these vertices are likely to be clustered near each other in the de Bruijn graph, as they likely come from a highly abundant transcript, or from a sequence that is found in multiple transcripts. Thus, if we use these nearby vertices as seeds, the contigs will collide early on and the resulting contigs will be very small. This can also be a problem for our parallel algorithm. If there are too many threads operating in the de Bruijn graph, there will be threads running simultaneously which have seeds close to one another. We will explore the effect that the number of parallel threads has on assembly quality and run time in [section 4.4](#).

3.1.3 Evaluation of Connections Between Contigs

After the greedy contig construction, the edges connecting contigs are evaluated. Each such edge is extended by $(k-1)/2$ onto both contigs, producing a sequence of length $2*k-1$, referred to as a “weld”. A weld is considered valid if it appears in the read set at least 4% as much as the average $(k-1)$ -mer of each of the contigs it connects. This stage corresponds to the Chrysalis stage of the Trinity algorithm.

Because all the welds are the same length, searching for the welds in the read set is well suited for the Rabin-Karp [12] algorithm. In the Rabin-Karp Algorithm, an integer hash is calculated for the weld with a polynomial hash function. Each character in the alphabet is assigned an integer between 0 and $A-1$, where A is the size of the alphabet. In our case, we use the 2-bit encoding of nucleotides, so $A = 4$. A large prime p is also chosen. Then, starting with the rightmost character of the weld and proceeding left, the key is calculated as follows:

```
1 for (i = len(weld); i > 0; i--) {
2   c = integer encoding of character i
3   hash = (hash * A + c) % p
4 }
```

Because a polynomial hash function is used, a corresponding hash can be calculated for each position in the read set in linear time. First one must calculate the hash represented by the first b characters of the read, where $m = 2*k-1$, the length of the weld. To get the hash of the next position, add

the next character entering the rolling window and subtract the one leaving.

$$hash = ((hash * A) + (new_c) - A^{m-1} * old_c) \% p$$

When you encounter a hash value in the text that matches the weld’s hash value, you must check that position for a match character by character. Because this checking could theoretically occur at every position in the text, the algorithm takes $O(N * m)$ time, where N is the total length of the read set.

To adapt the algorithm for multiple weld matching, we used a hash table with open addressing and stored pointers to the corresponding weld. When a hash from the a text hit an entry in the hash table, it checks that entry (and possibly more) for a match. It should be noted that the algorithm cannot be as easily adapted for multiple pattern matching if the patterns have different lengths.

The parallel implementation is straightforward. Each thread is first assigned to a weld, which it processes and adds to the hash table, resolving collisions if necessary. Then each thread is assigned to a read and processes it independently from the other threads.

3.1.4 Graph Simplification

In this stage we used heuristics inspired by the SOAPdenovo-Trans and Oases algorithms. We iterate through three steps until no further simplifica-

tion is performed: we compact the graph as described in section 3.1.1.4, we remove “short tips”, and we merge “bubbles”.

3.1.4.1 Short Tip Removal

A short tip is any single vertex v which represents a sequence of length $\leq 2 * k$ and has either no predecessor or no successor. Assume for the case of explanation that the vertex has no successor. Then we inspect its predecessor p . If p has a successor with a higher count than v , or which continues longer than v , we remove v .

The reasoning behind this heuristic is that short tips branching off an otherwise linear nucleotide sequence can be generated by a sequencing error within $k-1$ nucleotides of the end of a read. The error rate for Illumina is around 2%, so we expect that most short tips in the de Bruijn graph are produced by errors.

3.1.4.2 Bubble Merging

A bubble is a structure in the de Bruijn graph where there are two paths which start at some vertex u and end at some vertex v , without sharing any vertices between u and v . Like short tips, these structures are often caused by sequencing errors, this time near the middle of the read. For example, if there is an error in one nucleotide of one read, we expect this to generate a bubble of $2 * k - 2$ nucleotides. This is because there will be $k - 1$ $(k-1)$ -mers

overlapping the error, before returning to the current sequence.

In our algorithm we remove the majority of bubbles by inspecting every vertex v with a single predecessor u and a single successor w . We search for alternate paths between u and w , and if one of these paths has a greater count than v and has at least 90% identity with v 's sequence, we remove v and merge it into the alternate path. See figure 3.3 on page 26 for an example of a short tip and a bubble.

3.1.5 Reconstruction of Full Length Transcripts

After the de Bruijn graph has been simplified, the final stage is to find paths through the graph which are supported by reads. These nucleotide sequences of these paths are reported as transcripts in the final output of the assembler. This corresponds to the Butterfly stage of Trinity. At this point in the algorithm, there are many connected components consisting of a single vertex. These represent sequencing for which there is branching or other ambiguity, and can be reported as transcripts if they are long enough.

There are also connected components which contain forks, bubbles, and more complicated structures. These components can result from sequencing errors that were not resolved using the heuristics of the previous stages, they may be caused by alternative splicing during messenger RNA transcription, or they may be the result of genes which share sequences of length $\geq k - 1$. See figure 3.4 on page 27.

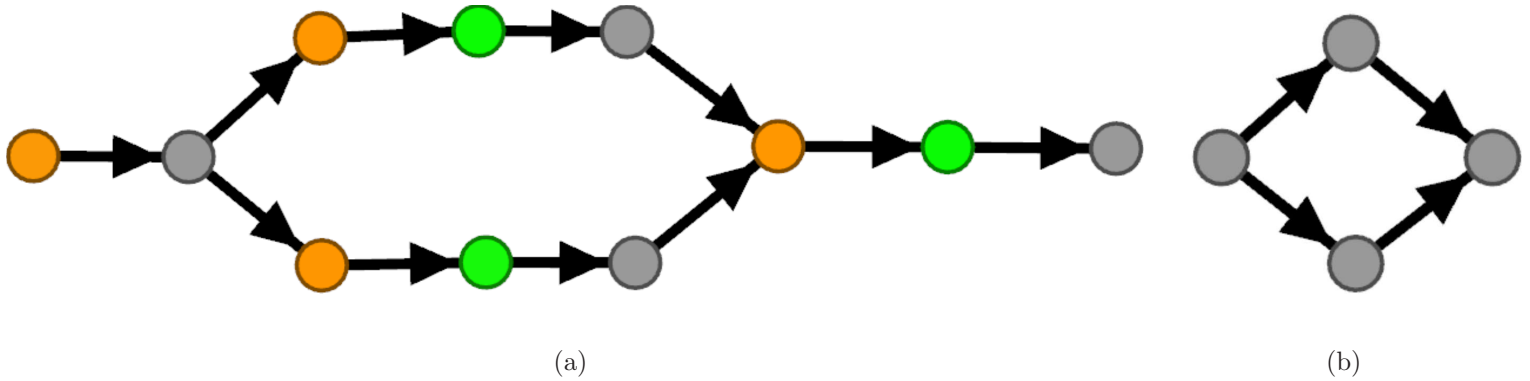


Figure 3.2: A graph (a) before and (b) after compaction. Both orange and green vertices are considered “linear” because they have only one successor, and that successor has only one predecessor. Orange vertices are also the first of a linear sequence, because their predecessor (or predecessors) are not linear. Each thread starts at an orange vertex and merges subsequent vertices until it reaches a non-linear vertex, which is also merged.

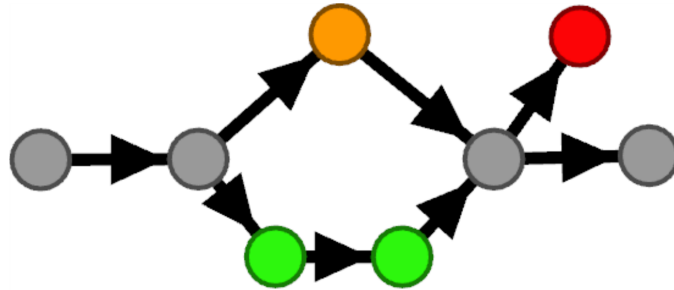


Figure 3.3: A graph with a bubble and a short tip. The bubble consists of the orange and green vertices. If the average count of the green path is greater than that of the orange vertex, and the sequence of the green path is has at least 90% identity with the orange vertex, the orange vertex is removed. The red vertex is a short tip, assuming its sequence has $\leq 2 * k$ nucleotides. If its sibling vertex has a higher average count or a longer sequence, the red vertex is removed.

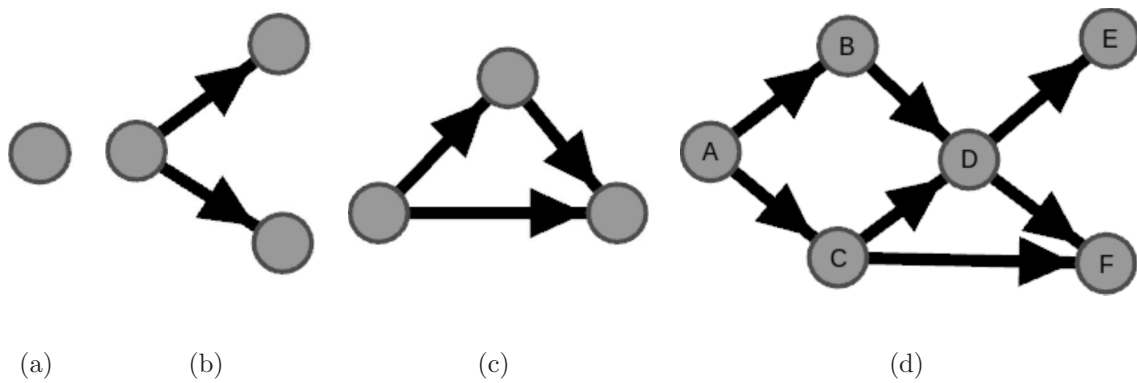


Figure 3.4: Some typical structures of connected components during the transcript reconstruction stage:

- (a) A single vertex: this represents a linear sequence, any sequencing errors were resolved in earlier stages.
- (b) A fork: this may represent a sequencing error or a pair of transcripts which share one or more introns before diverging.
- (c) A bubble: this may represent a sequencing error or a pair of transcripts, one of which has an extra intron.
- (d) More complex structures: these can arise by a combination of sequencing errors, alternative splicing, and multiple genes which share sequences of length k or longer.

We attempt to resolve which paths in the de Bruijn graph represent true transcripts by using paired reads. Up to this point, we have only considered reads individually, and have only considered the k -mer profile of the read set (except when evaluating welds, where we looked at $(2*k-1)$ -mers). Individual reads are typically less than 100 nucleotides long, but these reads are taken from RNA fragments of up to 500 nucleotides. In our datasets, the reads are 70 nucleotides long, and the average fragment length is 250 nucleotides with a standard deviation of 25 nucleotides.

We can determine the likelihood of paths by checking if there are paired reads consistent with every subpath of the average fragment length. We will describe how this is done in the following sections, but in the next two paragraphs we will describe a limitation of this approach.

Unfortunately, we cannot resolve which alternative paths are not consistent with each other if the distance between branches in the de Bruijn graph is greater than the average fragment length. Consider figure 3.4(d) on page 27. Assume that the algorithm determines that the subpaths $A-B-D$, $A-C-D$, $D-E$, and $D-F$ are all supported by paired reads, but no path including the edge $C-F$ is. If the sequence of D is smaller than the average than the average fragment length, then we can resolve which direction each of the paths ending in D continues by checking which subpaths spanning D are supported by paired reads. For example, if only $B-D-F$ and $C-D-E$ are supported, then the likely transcripts for this component include $A-B-D-F$ and $A-C-D-E$, but not $A-B-D-E$ or $A-C-D-F$.

However, if the sequence of D is larger than the average fragment length, then there will be few or no paired reads which span D. Thus we cannot resolve which combinations of paths are unlikely, and we must report all four possible paths. This limitation is a consequence of the fragment length and read length limitation during sequencing.

3.1.5.1 Read Mapping and Loop Resolution

Throughout the previous stages of the algorithm, the vertex and position within that vertex's sequence where each read starts is tracked. Now, we assign each read a thread, and track its path through the de Bruijn graph. In this way, we convert each read from a sequence of nucleotides into a list of vertices.

We now use reads to resolve short loops in the de Bruijn graph: self loops and double loops.

Self Loops:

Self loops are vertices are their own successor. This can be the result of a short repeated sequence in the transcriptome. We search the reads to find the maximum number of times each self loop vertices appears in a row. We then expand each self loop vertex by duplicating its sequence the maximum number of times that the self loop vertex was found in any read.

Double Loops:

A double loop is a vertex v which has only predecessor and one successor, and these are the same vertex, u . Once again, this can be the result of a short repeated sequence. We search the reads to find the maximum number of times v occurs in sequence with u . We remove v and expand the sequence of u by adding the sequence of v and u the maximum number of times that the double loop was found in any read.

3.1.5.2 Paired Read Extension

At this point we are done modifying the de Bruijn graph, and move on to considering paired reads together. We want to use paired reads to resolve longer sequences, up to the total length of the fragment the reads are derived from. If there is only one path (with a length less than the maximum fragment size) from the end of the first read to the beginning of the second read, we say that this pair of reads supports that entire path, and the two reads can be merged into single longer read.

If there are multiple such paths between the ends of a paired read, we will still attempt to extend the ends toward each other. We will extend the first end as long as the last vertex of the first end has only one successor which lies on all paths to the second end. When this extension is complete, we extend the second end back to the first end in the same manner. We then track the two ends together as a paired read. See figure [3.5](#) on page [32](#)

If there are no paths between the reads, they are tracked singly. If a

read consists of only a single vertex, it is discarded as it won't help resolve alternative paths.

Building a Hash Table of Distances Between Vertices:

In order to extend the reads, we need to know whether there is a path between any two vertices shorter than the maximum fragment size. So, we build a hash table- the key is the pair of vertices, and the value is the distance between the two vertices, including the end vertices. The hash table will only hold pairs of vertices for which that distance is smaller than the maximum fragment size.

To do this, we assign a thread to each vertex u , and use Dijkstra's shortest path algorithm [3]. When we find the shortest distance to a vertex v , we add the pair $u-v$ to the hash table. When we exceed the maximum fragment size, we stop early. Once this hash table is built, we can use it to check whether a path exists between a pair of vertices, as is needed to extend the reads.

3.1.5.3 Paired Read Grouping

After extending paired reads, there are many identical reads, and we would like to choose a representative read for each group of identical reads so we have fewer reads to track in the next stage. To do this, we build a "read forest" as follows:

- Each vertex in the de Bruijn graph is assigned a vertex in the read forest, and this vertex is the root of a tree.

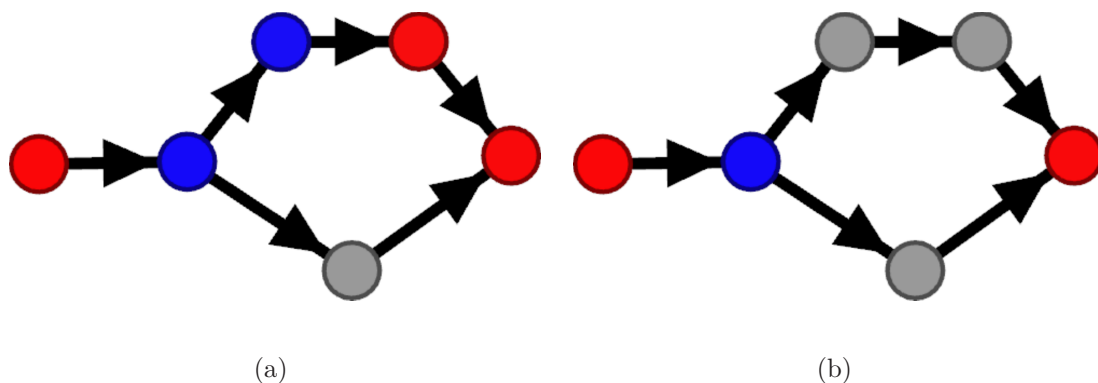


Figure 3.5: Examples of paired read extension (red vertices represent the original vertices of the paired reads, and blue vertices represent the vertices added after extension):

- (a) The ends of the paired read have only one path connecting them, so they are merged into a single read.
- (b) The ends of the paired read have multiple paths and therefore cannot be merged into a single read. However, the first end can be extended by one vertex, since all paths between the ends of the reads use the blue vertex.

- Each thread is assigned a read. We follow the vertex sequence of the read and descend the read forest by one vertex for every vertex (or gap between paired ends) in the read. If the corresponding vertex in the read forest does not yet exist, it is created.
- Once we reach the final vertex of the read in the de Bruijn graph, we have reached the vertex in the read forest which represents the unique path of that read. We check if a representative read has already been assigned for this unique path. If so, we increment the count of that representative read and mark the current read for deletion. If not, the current read is set as the representative read for that path.

For each vertex in the de Bruijn graph, we also build a list of the representative reads which start at that vertex.

3.1.5.4 Transcript Building

At this point in the algorithm, the de Bruijn graph has been contracted and simplified using only unpaired read information, and the paired reads themselves have been represented as lists of vertices in the graph, extended, and grouped. All that remains is to determine which paths through the de Bruijn graph are supported by the paired reads, and to report the sequence of these paths as assembled transcripts.

We start by assigning a thread to each source vertex (a vertex with no predecessor). This thread will keep track of two things: the path it traverses

as it builds a transcript, and a list of reads which are compatible with this path so far.

The thread then builds a transcript by checking each of the successors of the last vertex in its path. If none of the possible extensions of the path is supported by its reads, the thread is done building its transcript path. If exactly one of the possible extensions is supported, the thread will add the next vertex to its path and continue checking the extensions of that vertex. If more than one of the possible extensions is supported, the thread must copy the information it is tracking (path so far and compatible reads), and spawn new threads for every supported extension beyond one. The thread and all the newly spawned threads will continue checking their vertices for extension.

Eventually each thread will reach a vertex with no supported extension, or a vertex that is already in its path, and stop building its path. Once this happens, we find the vertices in the graph which have not been visited, and which have no predecessors which have not been visited. These are our new source vertices, and we repeat the transcript building process. This continues until all vertices have been visited.

Determining if a Path Extension is Supported by Reads:

When deciding whether to extend a path to a new vertex, we check whether some number of reads support the extension (2 by default). We say that a read supports an extension if it contains both the new vertex and the vertex some number of nucleotides back in the path. This number should be

the expected distance between the ends of a paired read (before extension). In this manner, we are able to use information from the full length of the sequenced fragments to resolve branches in the de Bruijn graph.

When an extension is made, the algorithm checks the tracked reads to make sure they are still compatible and relevant. If the read diverges from the path, or if we have passed the first end into the gap between the ends and have gone too far to rejoin the second end, we stop tracking that read for this transcript path.

Final Output

Once all the transcripts have been built, we filter out those that are too short and print the remaining transcripts in the FASTA format.

3.2 Evaluation of the XMT Assembler

3.2.1 Datasets

For each dataset, we chose a number of transcripts from chromosome 22, as annotated in the NCBI Reference Sequence Database [21]. These transcripts represent the “ground truth”, so the output of the transcriptome assembly algorithm should be as close to these transcripts as possible. The number of transcripts used were 1, 2, 5, 10, 20, 50, and 100.

We then used the polyester [5] package of bioconductor to simulate a set of reads. Polyester simulates the steps of RNA sequencing by creating

fragments with the expected distribution of fragment sizes and sequencing both ends of those fragments to create paired reads. It also introduces errors based on an error model for the sequencing platform being simulated. For our datasets, we simulated strand specific paired reads of 70 nucleotides, from fragments with a mean length of 250 and standard deviation of 25. We used the 'illumina5' error model. For each set of transcripts, we produced two sets of reads- one where each transcript had an average coverage of 20, and one where each read had a different average coverage between 0 and 40. Coverage is defined as the total number of nucleotides in the reads derived from a transcript divided by the number of nucleotides in that transcript.

3.2.2 Assemblers

We assembled each datasets using the XMT assembler and three other assemblers:

- Oases: Assemblies were created using all odd k-mer sizes between 20 and 32, and then merged into a final assembly using a k-mer size of 25. We chose the strand-specific option, indicated the average fragment length of 250 nucleotides, and reported only transcripts longer than 200 nucleotides.
- SOAPdenovo-Trans: Assemblies were created using the default k-mer size of 23. Again we chose the strand-specific option and indicated the average fragment length of 250 nucleotides.

- Trinity: Assemblies were created using the default k-mer size of 25. We chose the strand-specific option and used the default maximum fragment length of 500 nucleotides.
- XMT Assembler: The XMT architecture was simulated using XMTsim [13]. The configuration used for most testing had 1024 TCUs arranged in 64 clusters of 16 TCUs each, and connected to 128 cache modules using a mesh of trees interconnection network

3.2.3 Evaluation Metrics

3.2.3.1 Run Time and Memory Use

All of the assemblers besides the XMT assembler were run on an Intel Core i7-4810MQ processor with a clock frequency of 2.80 GHz. We used the unix time utility to measure the total processor time for each assembly (user and system time). To measure the peak memory use of the assemblers, we used the valgrind massif memory profiler [20].

To measure the run time and memory use of the XMT assembler, we ran the assembler on XMTsim. XMTsim is a cycle-accurate simulator for the XMT architecture, and can be used to measure clock cycles and total memory use. In this paper we convert the clock cycle count to the wall-clock time of an XMT computer using a clock frequency of 2.80 GHz, for direct comparison to the run time of the other assemblers.

3.2.3.2 Assembly Quality

Transcript Length Metrics:

The sequence length profile is a metric commonly used to evaluate the quality of genome assemblies. It is less appropriate for evaluating the quality of a transcriptome assembly, because transcriptome assemblies are expected to be more fragmented. However, we collected sequence length statistics from our assemblies in order to characterize the assemblies.

- Number of transcripts
- Length of longest transcript
- N50, N90: The N50 statistic is the length of the shortest transcript in the smallest subset of transcripts which contains at least 50% of the total number of nucleotides in the assembly. N90 is defined similarly.
- L50, L90: The L50 statistic is the least number of transcripts which contain at least 50% of the total number of nucleotides in the assembly.

Sensitivity and Specificity:

Since we use simulated datasets, we have a “ground truth” set of transcripts to compare our assemblies to. We use the BLAT aligner [15] to find alignments between the assembled transcripts and true transcripts which have a similarity of 95% or greater. Using these alignments we calculate the sensitivity and specificity of the assembly.

- Sensitivity: This is the proportion of true transcripts which can be aligned to an assembled transcript with at least 95% similarity.
- Specificity: This is the proportion of assembled transcripts which can be aligned to a true transcript with at least 95% similarity.

Log Average Probability (LAP):

Another approach to assessing the quality of an assembly is one which does not use the “ground truth” set of transcripts. This method, described in [6], instead calculates the probability that each individual read in the read set would be produced by a transcriptome sequencer, assuming that the assembly is correct. The assembly quality can then be defined as the geometric mean of each individual read probability. The log of this value is called the LAP, and is equal to the arithmetic mean of the log of individual read probabilities. We use the program, also called LAP, to calculate the LAP statistic for each assembly.

Chapter 4: Results

4.1 Assembly Quality

We found that the XMT Assembler produced results that were qualitatively similar to the SOAPdenovo-Trans results, in terms of transcript length statistics. Trinity typically assembles more transcripts than the XMT Assembler of SOAPdenovo-Trans, especially for larger datasets. In the largest datasets, the XMT Assembler produces shorter transcripts than Trinity and SOAPdenovo-Trans. Oases consistently produced shorter and more transcripts (so more fragmented in general) than the other assemblers. The differences between length statistics for assemblies of the constant and variable coverage datasets were minor. See table 4.1 on page 41 for the length statistics on the 5, 20, and 100 transcript datasets.

The specificity of the XMT Assembler was consistently higher than that of any of the other assemblers, but the sensitivity was worse than Trinity and SOAPdenovo-Trans for the three largest datasets. Trinity had the highest sensitivity in general. The LAP for the XMT Assembler was higher than for all other assemblers besides Trinity, which had a higher LAP statistic than

Table 4.1: Transcript length statistics for data sets with 5, 20, and 100 “ground truth” transcripts. The unit is # nucleotides for all numbers besides the # transcripts column.

Data set (# Tran.)	Coverage	Assembler	# Tran. Assembled	Longest Tran.	N50	N90	L50	L90
5	Constant	trinity	4	2050	1688	742	2	4
		oases	7	926	648	399	3	6
		soap	7	2041	1424	735	2	4
		xmt	4	2050	1799	742	2	4
	Variable	trinity	4	2052	1705	724	2	4
		oases	10	1219	619	281	3	8
		soap	4	2050	1866	706	2	4
		xmt	6	2052	1538	724	2	4
20	Constant	trinity	20	8593	3060	1467	5	15
		oases	74	3060	892	361	18	55
		soap	18	8563	2700	1363	4	12
		xmt	18	8593	2711	1475	5	13
	Variable	trinity	19	8608	2709	1373	4	13
		oases	68	2511	894	304	17	51
		soap	16	8600	2642	1451	4	11
		xmt	15	8608	2709	1483	4	11
100	Constant	trinity	99	9837	4494	1727	28	74
		oases	316	2753	803	323	80	237
		soap	68	9847	3938	1536	16	45
		xmt	69	8087	3241	1450	17	50
	Variable	trinity	101	9836	4093	1703	27	70
		oases	244	3447	876	312	55	179
		soap	70	9452	4047	1451	16	45
		xmt	68	9835	2723	1432	18	48

the XMT Assembler for some datasets. The sensitivity, specificity, and LAP statistics for Oases were consistently much worse than for the other three assemblers. See table 4.2 on page 43 for the assembly quality statistics on the 5, 20, and 100 transcript datasets and figure 4.1 on page 44 for statistic comparison on all datasets with constant coverage.

4.2 Run Time and Memory Use

The fastest serial assembler for most datasets was SOAPdenovo-Trans, though Oases was faster for the smallest datasets. Trinity was slower than all other assemblers, by a factor of 5 or more. The XMT assembler was faster than all the serial assemblers on all datasets, with speedups of between 25x and 61x over the fastest serial assembler. See table 4.3 on page 45 for the run time and memory use of the assemblers on the 5, 20, and 100 transcript datasets and figure 4.1 on page 44 for run time and memory comparisons on all datasets with constant coverage. See figure 4.2 on page 47 to see the XMT Assembler’s speedup over other assemblers on all datasets.

4.3 Runtime Scaling with Processor Count

In order to evaluate how the total work done by the XMT Assembler compares to that of the other assemblers, we simulated a configuration of the XMT architecture which had only one TCU, so the algorithm was performed serially. We found the XMT assembler had a 2x slowdown over the fastest

Table 4.2: Assembly quality statistics for data sets with 5, 20, and 100 “ground truth” transcripts.

Data set (# Tran.)	Coverage	Assembler	Sensitivity	Specificity	LAP
5	Constant	trinity	0.80	1.00	-8.39
		oases	0.00	1.00	-12.63
		soap	0.80	1.00	-8.76
		xmt	0.80	1.00	-8.21
	Variable	trinity	0.80	1.00	-8.53
		oases	0.00	1.00	-13.16
		soap	0.60	1.00	-8.77
		xmt	0.60	1.00	-8.48
20	Constant	trinity	0.95	1.00	-8.60
		oases	0.00	0.97	-13.19
		soap	0.95	1.00	-8.84
		xmt	0.90	1.00	-8.71
	Variable	trinity	0.90	1.00	-8.65
		oases	0.00	0.96	-13.47
		soap	0.90	0.94	-8.81
		xmt	0.85	1.00	-8.79
100	Constant	trinity	0.92	0.95	-9.11
		oases	0.03	0.91	-15.20
		soap	0.83	0.93	-9.62
		xmt	0.71	0.99	-9.49
	Variable	trinity	0.81	0.94	-9.11
		oases	0.04	0.93	-15.65
		soap	0.75	0.91	-9.60
		xmt	0.75	0.99	-9.43

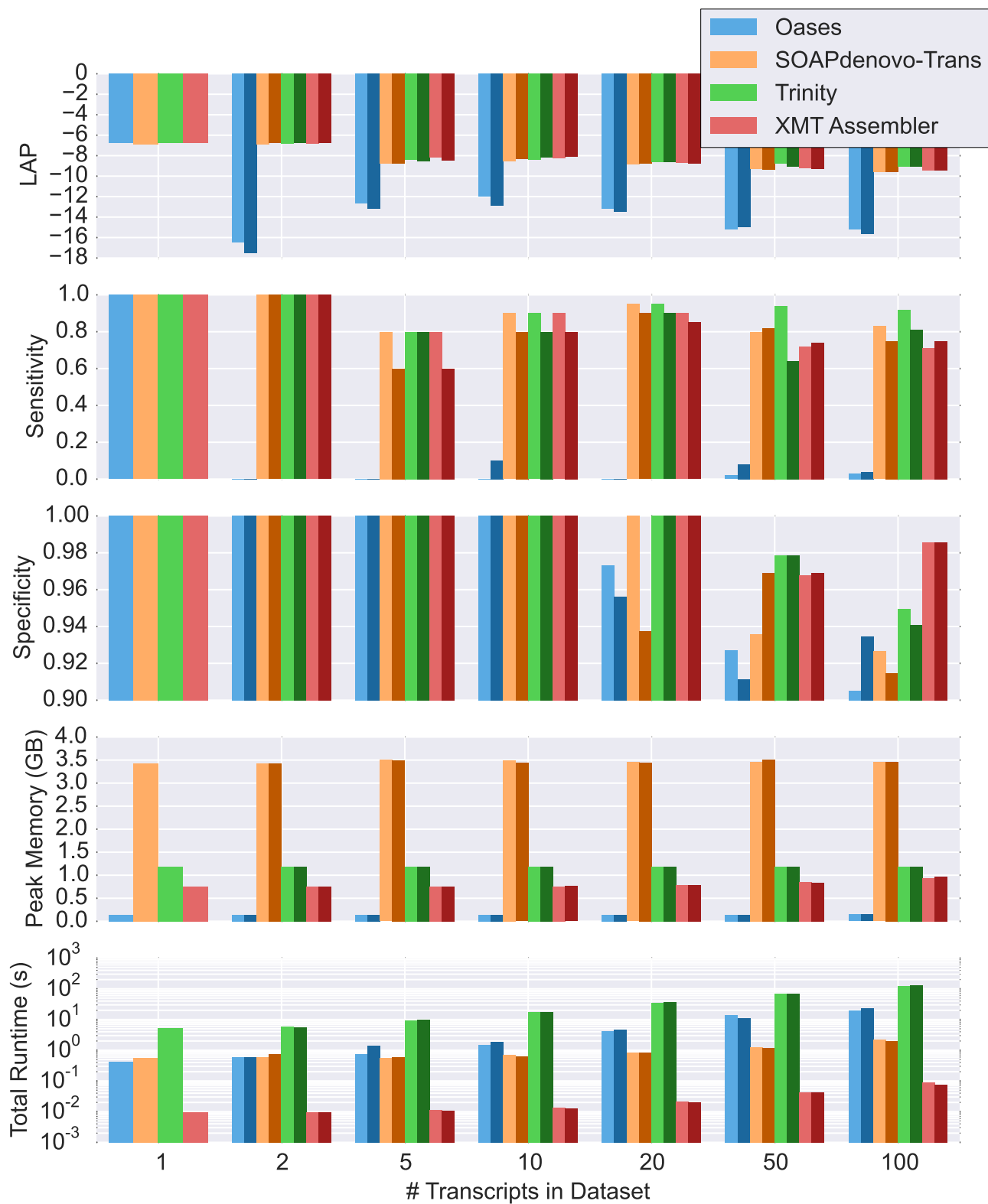


Figure 4.1: Comparison of evaluation metrics for assemblies of all datasets. The lighter bars represents datasets with constant coverage and the darker bars represent datasets with variable coverage.

Table 4.3: Total runtime and peak memory usage of assemblers for data sets with 5, 20, and 100 “ground truth” transcripts.

Data set (# Tran.)	Coverage	Assembler	Total Runtime (s)	Peak Memory (GB)
5	Constant	trinity	9.35	8.73
		oases	0.71	0.13
		soap	0.56	3.50
		xmt	0.011	0.75
	Variable	trinity	9.48	8.73
		oases	1.37	0.13
		soap	0.57	3.50
		xmt	0.011	0.76
20	Constant	trinity	34.74	8.73
		oases	3.97	0.14
		soap	0.83	3.45
		xmt	0.021	0.78
	Variable	trinity	36.54	8.73
		oases	4.51	0.14
		soap	0.83	3.44
		xmt	0.020	0.78
100	Constant	trinity	121.79	8.73
		oases	18.21	0.15
		soap	1.66	3.46
		xmt	0.089	0.94
	Variable	trinity	125.12	8.73
		oases	22.73	0.15
		soap	1.96	3.46
		xmt	0.074	0.96

serial assembler for the 100 transcript constant coverage dataset. We simulated XMT configurations with TCUs between 1 and 1024 (see figure 4.3 on page 47). There continues to be significant improvement in run time up to 128 TCUs, at which point adding more TCUs give diminishing returns.

4.4 Parallelism During Greedy Contig Construction

As mentioned in section 3.1.2, using more threads during the greedy contig construction stage of the XMT Assembler algorithm can result in a more fragmented graph and diminish the quality of the final assembly. For the datasets with 1, 10, and 100 transcripts and constant coverage, we used a number of threads between 1 and 1024 during contig construction. For all datasets, we see a sharp drop in LAP and sensitivity as we increase the number of threads. As we see in figure 4.4 on page 49, this drop occurs with a smaller number of threads for smaller datasets. This is as expected, because in the larger dataset the deBruijn graph is larger and parallel threads are less likely to collide with each other during contig construction. The specificity of the assemblies was relatively unaffected by the number of threads used during contig construction.

Figure 4.4 also shows the total run time of the algorithm as a function of the number of threads used during contig construction. Interestingly, after an initial drop, the run time increases with the number of threads. Although the run time for the contig construction stage decreases monotonically with the

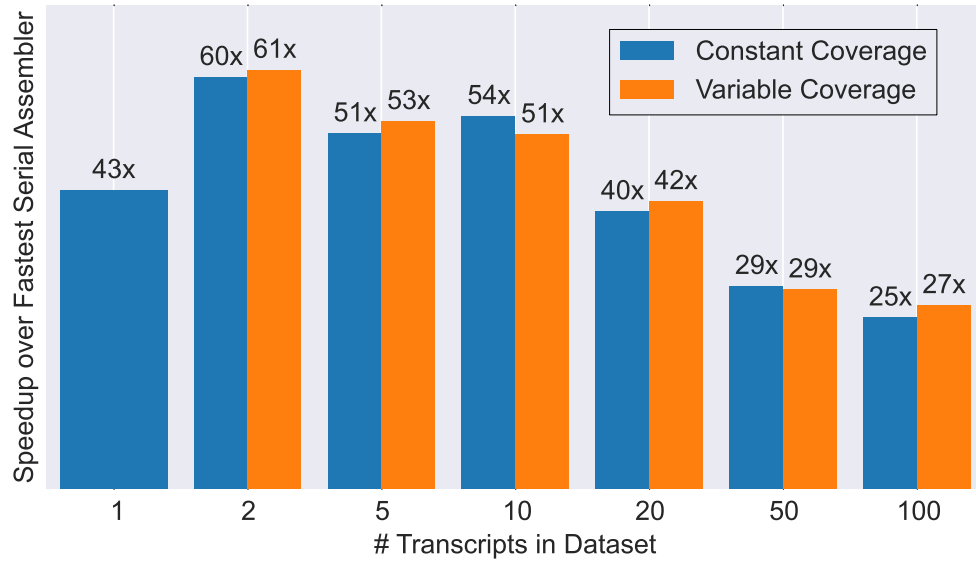


Figure 4.2: Speedup achieved by the XMT assembler over the fastest serial assembler for all datasets.

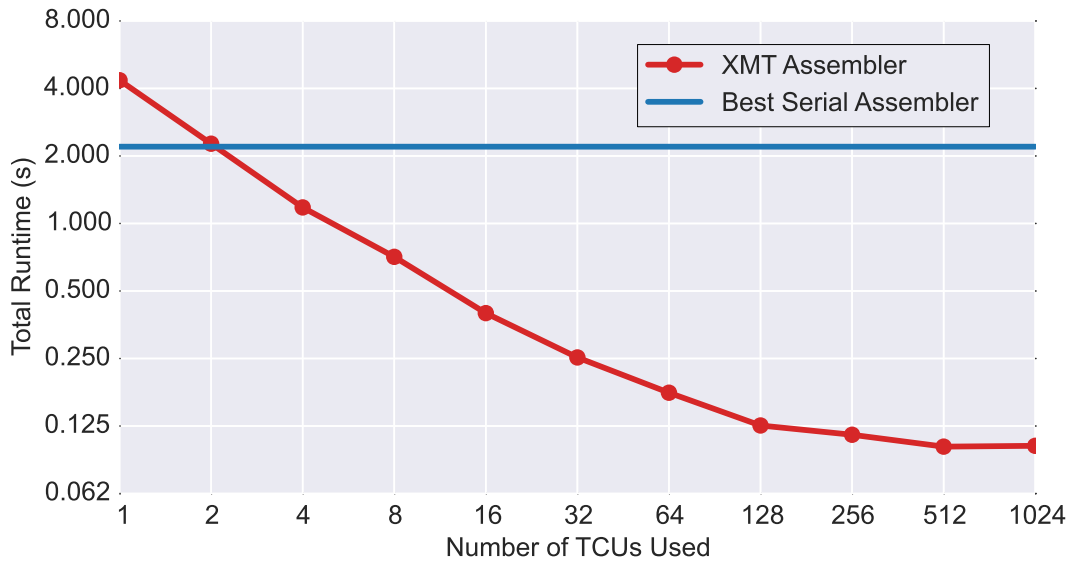


Figure 4.3: Effect of the number of TCUs on run time for the 100 transcript dataset with constant coverage.

number of threads, the run time of the next stage (evaluation of connections between contigs) increases. When there are enough threads that they begin to interfere during contig construction, the graph will be more fragmented and there will be more connections between contigs that need to be evaluated.

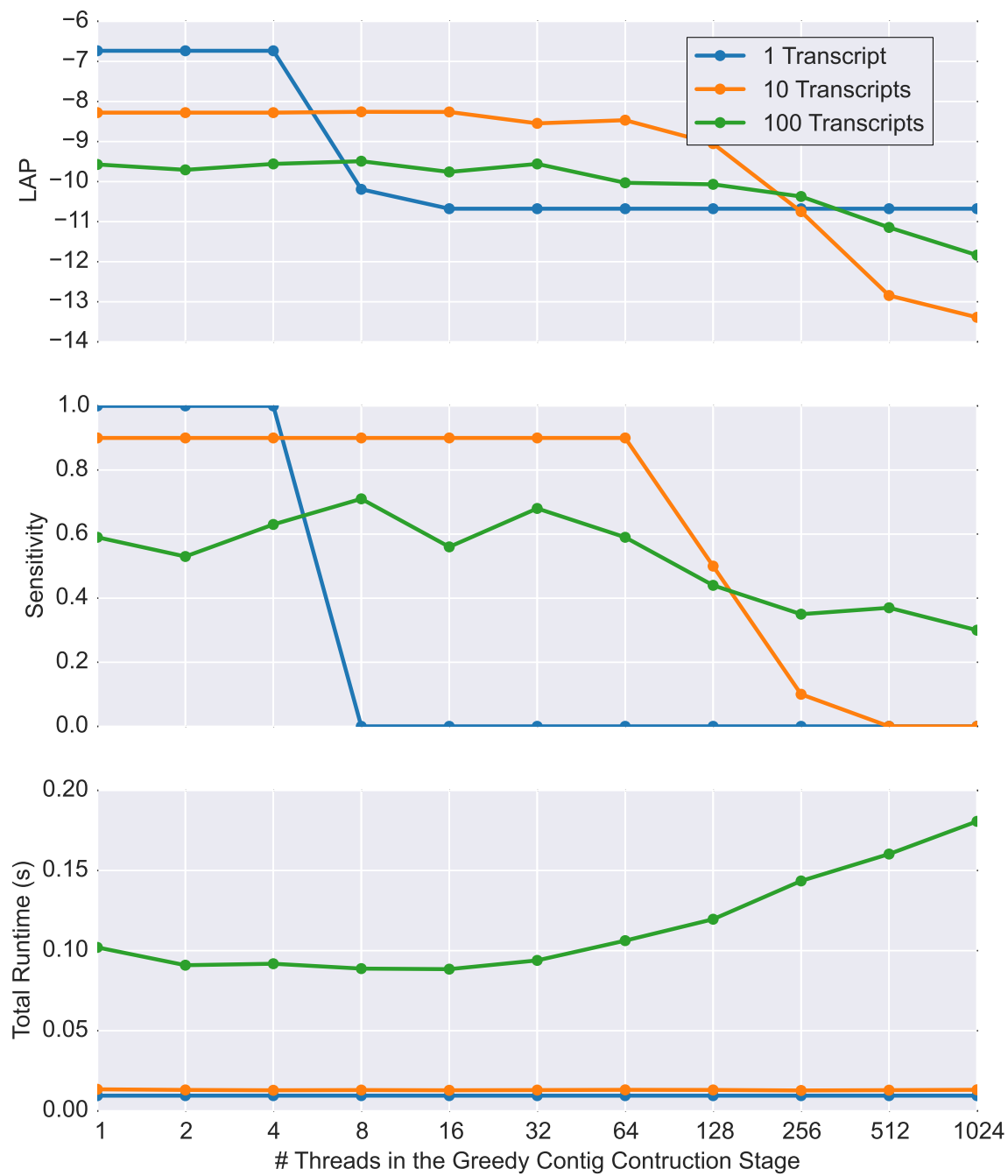


Figure 4.4: Effect of using more threads during the greedy contig construction stage on final assembly quality and speed. The constant coverage datasets were used.

Chapter 5: Conclusions

The scope of testing was limited by our use of the XMT simulator. The XMT simulator is limited to 2GB of memory and requires a large amount of wall-clock time to simulate a very short program. As a result, we could only test the XMT Assembler on very small datasets. We recommend more extensive testing of the XMT Assembler on large datasets to check if the promising speedup and quality results continue as the input size increases. This can be achieved if an updated version of the XMT simulator is made, or if a physical version of the XMT architecture is built.

The XMT Assembler achieved speedups of between 25x and 61x over the fastest serial de novo transcriptome assemblers tested, without falling behind other de novo transcriptome assemblers in assembly quality metrics. This brings the run time of de novo transcriptome assembly on par with genome-guided transcriptome assembly. Thus, the XMT Assembler can provide the benefits of de novo transcriptome assembly without the cost of greatly increased run time.

Bibliography

- [1] David R Bentley, Shankar Balasubramanian, Harold P Swerdlow, Geoffrey P Smith, John Milton, Clive G Brown, Kevin P Hall, Dirk J Evers, Colin L Barnes, Helen R Bignell, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *nature*, 456(7218):53–59, 2008.
- [2] Cancer Research UK. Diagram showing a double helix of a chromosome, July 2014.
- [3] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
- [4] James Edwards. personal communication.
- [5] Alyssa C. Frazee, Andrew E. Jaffe, Rory Kirchner, and Jeffrey T. Leek. *polyester: Simulate RNA-seq reads*, 2016. R package version 1.10.0.
- [6] Mohammadreza Ghodsi, Christopher M Hill, Irina Astrovskaya, Henry Lin, Dan D Sommer, Sergey Koren, and Mihai Pop. De novo likelihood-based measures for comparing genome assemblies. *BMC research notes*, 6(1):1, 2013.
- [7] Manfred G Grabherr, Brian J Haas, Moran Yassour, Joshua Z Levin, Dawn A Thompson, Ido Amit, Xian Adiconis, Lin Fan, Raktima Raychowdhury, Qiandong Zeng, Zehua Chen, Evan Mauceli, Nir Hacohen, Andreas Gnirke, Nicholas Rhind, Federica di Palma, Bruce W Birren, Chad Nusbaum, Kerstin Lindblad-Toh, Nir Friedman, and Aviv Regev. Full-length transcriptome assembly from RNA-Seq data without a reference genome. *Nat Biotechnol*, 29(7):644–652, July 2011.
- [8] Brenton R Graveley. Alternative splicing: increasing diversity in the proteomic world. *TRENDS in Genetics*, 17(2):100–107, 2001.

- [9] Robert Henschel, Matthias Lieber, Le-Shin Wu, Phillip M Nista, Brian J Haas, and Richard D LeDuc. Trinity RNA-Seq assembler performance optimization. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*, page 45. ACM, 2012.
- [10] François Jacob and Jacques Monod. Genetic regulatory mechanisms in the synthesis of proteins. *Journal of molecular biology*, 3(3):318–356, 1961.
- [11] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.
- [12] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [13] Fuat Keceli and Uzi Vishkin. XMTSim: A Simulator of the XMT Many-core Architecture. Technical report, University of Maryland, 2011.
- [14] Jörg Keller, Christoph Kessler, and Jesper Träff. *Practical PRAM programming*. Wiley-Interscience, J. Wiley & Sons, Inc., 2001.
- [15] W James Kent. BLAT—the BLAST-like alignment tool. *Genome research*, 12(4):656–664, 2002.
- [16] Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, Nan Li, Xuesong Hu, Binghang Liu, et al. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Briefings in functional genomics*, 11(1):25–37, 2012.
- [17] BingXin Lu, ZhenBing Zeng, and TieLiu Shi. Comparative study of de novo assembly and genome-guided assembly strategies for transcriptome reconstruction based on RNA-Seq. *Science China Life sciences*, 56(2):143–155, 2013.
- [18] Jeffrey A Martin and Zhong Wang. Next-generation transcriptome assembly. *Nature Reviews Genetics*, 12(10):671–682, 2011.
- [19] National Institutes of Health. Illustration of Transcription and Translation, October 2004.
- [20] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.
- [21] Kim Pruitt, Garth Brown, Tatiana Tatusova, and Donna Maglott. The Reference Sequence (RefSeq) Database. In Jo McEntyre and Jim Ostell, editors, *The NCBI Handbook*, chapter 18. National Center for Biotechnology Information, Bethesda, MD, October 2002.

- [22] Jorge S Reis-Filho. Next-generation sequencing. *Breast Cancer Research*, 11(3):1, 2009.
- [23] Gordon Robertson, Jacqueline Schein, Readman Chiu, Richard Corbett, Matthew Field, Shaun D Jackman, Karen Mungall, Sam Lee, Hisanaga Mark Okada, Jenny Q Qian, et al. De novo assembly and analysis of RNA-seq data. *Nature methods*, 7(11):909–912, 2010.
- [24] Martin Rosenberg and D Court. Regulatory sequences involved in the promotion and termination of RNA transcription. *Annual review of genetics*, 13(1):319–353, 1979.
- [25] Marcel H. Schulz, Daniel R. Zerbino, Martin Vingron, and Ewan Birney. Oases: robust de novo RNA-seq assembly across the dynamic range of expression levels. *Bioinformatics*, 28(8):1086–1092, 2012.
- [26] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. ABySS: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.
- [27] Uzi Vishkin. Using simple abstraction to reinvent computing for parallelism. *Commun. ACM*, 54(1):75–85, 2011.
- [28] James D Watson, Francis HC Crick, et al. Molecular structure of nucleic acids. *Nature*, 171(4356):737–738, 1953.
- [29] Yinlong Xie, Gengxiong Wu, Jingbo Tang, Ruibang Luo, Jordan Patterson, Shanlin Liu, Weihua Huang, Guangzhu He, Shengchang Gu, Shengkang Li, Xin Zhou, Tak Wah Lam, Yingrui Li, Xun Xu, Kane Ka-Shu Wong, and Jun Wang. SOAPdenovo-Trans: de novo transcriptome assembly with short RNA-Seq reads. *Bioinformatics*, 30(12):1660–1666, 2014.